Automatic Sanskrit Segmentizer Using Finite State Transducers

Vipul Mittal
Language Technologies Research Center, IIIT-H,
Gachibowli, Hyderabad, India.
vipulmittal@research.iiit.ac.in

Abstract

In this paper, we propose a novel method for automatic segmentation of a Sanskrit string into different words. The input for our segmentizer is a Sanskrit string either encoded as a Unicode string or as a Roman transliterated string and the output is a set of possible splits with weights associated with each of them. We followed two different approaches to segment a Sanskrit text using *sandhi*[1] rules extracted from a parallel corpus of manually sandhi split text. While the first approach augments the finite state transducer used to analyze Sanskrit morphology and traverse it to segment a word, the second approach generates all possible segmentations and validates each constituent using a morph analyzer.

1 Introduction

Sanskrit has a rich tradition of oral transmission of texts and this process causes the text to undergo euphonic changes at the word boundaries. In oral transmission, the text is predominantly spoken as a continuous speech. However, continuous speech makes the text ambiguous. To overcome this problem, there is also a tradition of reciting the pada-pāṭha (recitation of words) in addition to the recitation of a saṃhitā (a continuous sandhied text). In the written form, because of the dominance of oral transmission, the text is written as a continuous string of letters rather than a sequence of words. Thus, the Sanskrit texts consist of a very long sequence of phonemes, with the word boundaries having undergone euphonic changes. This makes it difficult to split a continuous string into words and process the text automatically.

Sanskrit words are mostly analyzed by building a finite state transducer (Beesley, 1998). In the first approach, this transducer was modified by linking the final states to appropriate intermediate states incorporating the sandhi rules. This approach then allows one to traverse the string from left to right and generate all and only possible splits that are morphologically valid. The second approach is very closely based on the *Optimality Theory* (Prince and Smolensky, 1993) where we generate all the possible splits for a word and validate each using a morphological analyzer. We use one of the fastest morphological analyzers available viz. the one developed by *Apertium* group[2]. The splits that are not validated are pruned out. Based on the number of times the first answer is correct, we achieved an accuracy of around 92% using the second approach while the first approach performed with around 71% accuracy.

2 Issues involved in Sanskrit Processing

The segmentizer is an important component of an NLP system. Especially, languages such as Chinese (Badino, 2004), Japanese, Thai (Haruechaiyasak, 2008) or Vietnamese (Thang et al. , 2008) which do not mark word boundaries explicitly or highly agglutinative languages like Turkish need segmentizers. In all these languages, there are no explicit delimiters to specify the word boundaries. In Thai, each syllable is transcribed using several characters and there is no space in the text between syllables. So the problem of segmentation is basically twofold: (1) syllable segmentation followed by (2) word segmentation itself. A sentence in these languages

[1] *Sandhi* means euphony transformation of words when they are consecutively pronounced. Typically when a word w_1 is followed by a word w_2, some terminal segment of w_1 merges with some initial segment of w_2 to be replaced by a "smoothed" phonetic interpolation, corresponding to minimizing the energy necessary to reconfigurate the vocal organs at the juncture between the words.

[2] http://wiki.apertium.org/wiki/lttoolbox; It processes around 50,000 words per sec.

Proceedings of the ACL 2010 Student Research Workshop, pages 85–90,
Uppsala, Sweden, 13 July 2010.

is segmented by predicting the word boundaries, where euphonic changes do not occur across the word boundaries and it is more like mere concatenation of words. So the task here is just to choose between various combinations of the words in a sentence.

However, in Sanskrit, euphonic changes occur across word boundaries leading to addition and deletion of some original part of the combining words. These euphonic changes in Sanskrit introduce non-determinism in the segmentation. This makes the segmentation process in Sanskrit more complex than in Chinese or Japanese. In case of highly agglutinative languages like Turkish, the components are related to each other semantically involving dependency analysis. Whereas in Sanskrit, only the compounds involve a certain level of dependency analysis, while sandhi is just gluing of words together, without the need for words to be related semantically. For example, consider the following part of a verse,

San: *nāradam paripapraccha vālmīkirmunipuṅgavam*
gloss: to_the_Narada asked Valmiki-to_the_wisest_among_sages
Eng: Valmiki asked the Narada, the wisest among the sages.

In the above verse, the words *vālmīkiḥ* and *munipuṅgavam* (wisest among the sages - an adjective of Narada) are not related semantically, but still undergo euphonic change and are glued together as *vālmīkirmunipuṅgavam*.

Further, the split need not be unique. Here is an example, where a string *māturājñāmparipālaya* may be decomposed in two different ways after undergoing euphonic changes across word boundaries.

- *mātuh ājñām paripālaya* (obey the order of mother) and,
- *mā āturājñām paripālaya* (do not obey the order of the diseased).

There are special cases where the sandhied forms are not necessarily written together. In such cases, the white space that physically marks the boundary of the words, logically refers to a single sandhied form. Thus, the white space is deceptive, and if treated as a word boundary, the morphological analyzer fails to recognize the word. For example, consider

śrutvā ca nārado vacaḥ.

In this example, the space between *śrutvā* and *ca* represent a proper word boundary and the word *śrutvā* is recognized by the morphological analyzer whereas the space between *nārado* and *vacaḥ* does not mark the word boundary making it deceptive. Because of the word *vacaḥ*, *nāradaḥ* has undergone a phonetic change and is rendered as *nārado*. In unsandhied form, it would be written as,

San: *śrutvā ca nāradaḥ vacaḥ.*
gloss: after_listening and Narada's speech
Eng: And after listening to Narada's speech

The third factor aggravating Sanskrit segmentation is productive compound formation. Unlike English, where either the components of a compound are written as distinct words or are separated by a hyphen, the components of compounds in Sanskrit are always written together. Moreover, before these components are joined, they undergo the euphonic changes. The components of a compound typically do not carry inflection or in other words they are the bound morphemes used only in compounds. This forces a need of a special module to recognize compounds.

Assuming that a sandhi handler to handle the sandhi involving spaces is available and a bound morpheme recognizer is available, we discuss the development of sandhi splitter or a segmentizer that splits a continuous string of letters into meaningful words. To illustrate this point, we give an example.
Consider the text,

śrutvā caitattrilokajño vālmīkernārado vacaḥ.

We assume that the sandhi handler handling the sandhi involving spaces is available and it splits the above string as,

śrutvā caitattrilokajñaḥ vālmīkernāradaḥ vacaḥ.

The sandhi splitter or segmentizer is supposed to split this into

śrutvā ca etat triloka-jñaḥ vālmīkeḥ nāradaḥ vacaḥ.

This presupposes the availability of rules corresponding to euphonic changes and a good coverage morphological analyzer that can also analyze the bound morphemes in compounds.

A segmentizer for Sanskrit developed by Huet (Huet, 2009), decorates the final states of its finite state transducer handling Sanskrit morphology with the possible sandhi rules. However, it is still not clear how one can prioritize various splits with this approach. Further, this system in current state demands some more work before the sandhi splitter of this system can be used as a standalone system allowing plugging in of different morphological analyzers. With a variety of morphological analyzers being developed by various researchers[3], at times with complementary abilities, it would be worth to experiment with various morphological analyzers for splitting a sandhied text. Hence, we thought of exploring other alternatives and present two approaches, both of which assume the existence of a good coverage morphological analyzer. Before we describe our approaches, we first define the scoring matrix used to prioritize various analyses followed by the baseline system.

3 Scoring Matrix

Just as in the case of any NLP systems, with the sandhi splitter being no exception, it is always desirable to produce the most likely output when a machine produces multiple outputs. To ensure that the correct output is not deeply buried down the pile of incorrect answers, it is natural to prioritize solutions based on some frequencies. A Parallel corpus of Sanskrit text in sandhied and sandhi split form is being developed as a part of the Consortium project in India. The corpus contains texts from various fields ranging from children stories, dramas to Ayurveda texts. Around 100K words of such a parallel corpus is available from which around 25,000 parallel strings of unsandhied and corresponding sandhied texts were extracted. The same corpus was also used to extract a total of 2650 sandhi rules including the cases of mere concatenation, and the frequency distribution of these sandhi rules. Each sandhi rule is a triple (x, y, z) where y is the last letter of the first primitive, z is the first letter of the second primitive, and x is the letter sequence created by euphonic combination. We define the estimated probability of the occurrence of a sandhi rule as follows:

Let R_i denote the i^{th} rule with f_{R_i} as the frequency of occurrence in the manually split parallel text. The probability of rule R_i is:

$$P_{R_i} = \frac{f_{R_i}}{\sum_{i=1}^{n} f_{R_i}}$$

where *n* denotes the total number of sandhi rules found in the corpus.

Let a word be split into a candidate S_j with k constituents as $< c_1, c_2, ..c_k >$ by applying $k-1$ sandhi rules $< R_1, R_2, ..R_{k-1} >$ in between the constituents. It should be noted here that the rules $R_1, ..R_{k-1}$ and the constituents $c_1, ..c_k$ are interdependent since a different rule sequence will result in a different constituents sequence. Also, except c_1 and c_k, all intermediate constituents take part in two segmentations, one as the right word and one as the left.

The weight of the split S_j is defined as:

$$W_{S_j} = \frac{\prod_{x=1}^{k-1}(P_{c_x} + P_{c_{x+1}}) * P_{R_x}}{k}$$

where P_{c_x} is the probability of occurrence of the word c_x in the corpus. The factor of *k* was introduced to give more preference to the split with less number of segments than the one with more segments.

4 Baseline System

We define our own baseline system which assumes that each Sanskrit word can be segmented only in two constituents. A word is traversed from left to right and is segmented by applying the first applicable rule provided both the constituents are valid morphs. Using the 2,650 rules, on a test data of 2,510 words parallel corpus, the baseline performance of the system was around 52.7% where the first answer was correct.

5 Two Approaches

We now present the two approaches we explored for sandhi splitting.

5.1 Augmenting FST with Sandhi rules

In this approach, we build an FST, using OpenFst (Allauzen et al., 2007) toolkit, incorporating

[3]http://sanskrit.uohyd.ernet.in, http://www.sanskritlibrary.org, http://sanskrit.jnu.ernet.in

sandhi rules in the FST itself and traverse it to find the sandhi splittings.

We illustrate the augmentation of a sandhi rule with an example. Let the two strings be *xaXi* (dadhi)[4] and *awra* (atra). The initial FST without considering any sandhi rules is shown in Figure 1.

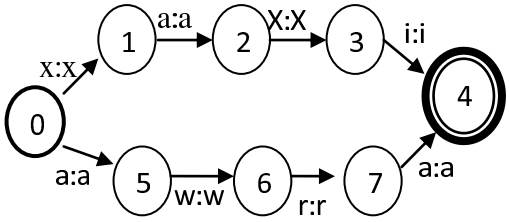

Figure 1: Initial FST accepting only two words *xaXi* and *awra*.

As the figure depicts, 0 is the start state and 4 is the final state. Each transition is a 4-tuple $<c, n, i, o>$ where c is current state, n is the next state, i is the input symbol and o is the output. The FST marks word boundaries by flushing out certain features about the words whenever it encounters a valid word. Multiple features are separated by a '|'. E.g., the output for *xaXi* is *lc,s*|*vc,s* and for *awra* it is *vc,s* where *lc,s* stands for *locative, singular* and *vc,s* is *vocative, singular*. The FST in Figure 1 recognize exactly two words *xaXi* and *awra*.

One of the sandhi rule states that *i*+*a* → *ya* which will be represented as a triple (ya, i, a). Applying the sandhi rule, we get: *xaX**i*** + ***a**wra* → *xaX**ya**wra*. After adding this sandhi rule to the FST, we get the modified FST that is represented in Figure 2.

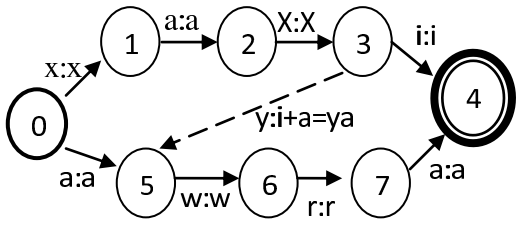

Figure 2: Modified FST after inserting the rule. − − − indicates the newly added transition.

Here, a transition arc is added depicting the rule which says that on receiving an input symbol *ya* at state 3, go to state 5 with an output *i*+*a* → *ya*. Thus the new FST accepts *xaXyawra* in addition to *xaXi* and *awra*.

Thus, we see that the original transducer gets modified with all possible transitions at the end of a final phoneme, and hence, also explodes the number of transitions leading to a complex transducer.

The basic outline of the algorithm to split the given string into sub-strings is:

Algorithm 1 To split a string into sub-strings

1: Let the FST for morphology be *f*.
2: Add sandhi rules to the final states of *f1* linking them to the intermediary states to get f'.
3: Traverse f' to find all possible splits for a word. If a sandhi rule is encountered, split the word and continue with the remaining part.
4: Calculate the weights of the possible outputs with the formula discussed in section 3.

The pseudo-code of the algorithm used to insert sandhi rules in the FST is illustrated here:

Algorithm 2 To insert sandhi rules in the FST

```
1: I = Input Symbol; X = last character of the result of the rule.
2: for each transition in the FST transition table do
3:    if next state is a final state then
4:       for all rules where I is the last character of first word do
5:          S = next state from the start state on encountering X;
6:          Y = first character of the result of the rule;
7:          transition T = current state, S, Y, rule;
8:          Add T into the FST;
9:       end for
10:   end if
11: end for
```

The main problem with this approach is that every finite state can have as many transitions as the number of euphonic rules resulting in phoneme change. This increases the size of the FST considerably. It should be noted that, we have not included the cases, where there is just a concatenation. In such cases, if the input string is not exhausted, but the current state is a final state, we go back to the start state with the remaining string as the input.

[4] A Roman transliteration scheme called WX transliteration is used, which is one-to-one phoneme level representation of Devanāgarī script.

5.1.1 Results

The performance of this system measured in terms of the number of times the highest ranked segmentation is correct, with around 500 sandhi rules, and only noun morphology tested on the same test data used for testing baseline system gave the following rank-wise distribution presented in Table 1.

Rank	% of output
1	71.2509
2	5.64543
3	3.85324
4	3.35651
5	1.56123
>5	14.33268

Table 1: Rank-wise Distribution for Approach-1.

The system was slow consuming, on an average, around 10 seconds per string of 15 letters.[5].

With the increase in the sandhi rules, though system's performance was better, it slowed down the system further. Moreover, this was tested only with the inflection morphology of nouns. The verb inflection morphology and the derivational morphology were not used at all. Since, the system is supposed to be part of a real time application viz. machine translation, we decided to explore other possibilities.

5.2 Approach based on Optimality Theory

Our second approach follows optimality theory(OT) which proposes that the observed forms of a language are a result of the interaction between the conflicting constraints. The three basic components of the theory are:

1. GEN - generates all possible outputs, or candidates.

2. CON - provides the criteria and the constraints that will be used to decide between candidates.

3. EVAL - chooses the optimal candidate based on the conflicts on the constraints.

OT assumes that these components are universal and the grammars differ in the way they rank the universal constraint set, CON. The grammar of each language ranks the constraints in some dominance order in such a way that every constraint must have outperformed every lower ranked constraint. Thus a candidate A is optimal if it performs better than some other candidate B on a higher ranking constraint even if A has more violations of a lower ranked constraint than B.

The GEN function produces every possible segmentation by applying the rules wherever applicable. The rules tokenize the input surface form into individual constituents. This might contain some insignificant words that will be eventually pruned out using the morphological analyser in the EVAL function thus leaving the winning candidate. Therefore, the approach followed is very closely based on optimality theory. The morph analyser has no role in the generation of the candidates but only during their validation thus composing the back-end of the segmentizer. In original OT, the winning candidate need not satisfy all the constraints but it must outperform all the other candidates on some higher ranked constraint. While in our scenario, the winning candidate must satisfy all the constraints and therefore there could be more than one winning candidates.

Currently we are applying only two constraints. We are planning to introduce some more constraints. The constraints applied are:

- C1 : All the constituents of a split must be valid morphs.

- C2 : Select the split with maximum weight, as defined in section 3.

The basic outline of the algorithm is:

1: Recursively break a word at every possible position applying a sandhi rule and generate all possible candidates for the input.
2: Pass the constituents of all the candidates through the morph analyzer.
3: Declare the candidate as a valid candidate, if all its constituents are recognized by the morphological analyzer.
4: Assign weights to the accepted candidates and sort them based on the weights.
5: The optimal solution will be the one with the highest salience.

5.2.1 Results

The current morphological analyzer can recognize around 140 million words. Using the 2650 rules

[5]Tested on a system with 2.93GHz Core 2 Duo processor and 2GB RAM

and the same test data used for previous approach, we obtained the following results:

- Almost 93% of the times, the highest ranked segmentation is correct. And in almost 98% of the cases, the correct split was among the top 3 possible splits.
- The system consumes around 0.04 seconds per string of 15 letters on an average.

The complete rank wise distribution is given in Table 2.

	% of output	
Rank	**Approach-1**	**Approach-2**
1	71.2509	**92.8771**
2	5.64543	5.44693
3	3.85324	1.07076
4	3.35651	0.41899
5	1.56123	0.09311
>5	14.33268	0.0931

Table 2: Complete rank-wise Distribution.

6 Conclusion

We presented two methods to automatically segment a Sanskrit word into its morphologically valid constituents. Though both the approaches outperformed the baseline system, the approach that is close to optimality theory gives better results both in terms of time consumption and segmentations. The results are encouraging. But the real test of this system will be when it is integrated with some real application such as a machine translation system. This sandhi splitter being modular, wherein one can plug in different morphological analyzer and different set of sandhi rules, the splitter can also be used for segmentization of other languages.

Future Work The major task would be to explore ways to shift rank 2 and rank 3 segmentations more towards rank 1. We are also exploring the possibility of including some semantic information about the words while defining weights. The sandhi with white spaces also needs to be handled.

Acknowledgments

I would like to express my gratitude to Amba Kulkarni and Rajeev Sangal for their guidance and support.

References

Akshar Bharati, Amba P. Kulkarni, and V Sheeba. 2006. *Building a wide coverage Sanskrit morphological analyzer: A practical approach*. The First National Symposium on Modelling and Shallow Parsing of Indian Languages, IIT-Bombay.

Alan Prince and Paul Smolensky. 1993. *Optimality Theory: Constraint Interaction in Generative Grammar*. RuCCS Technical Report 2 at Center for Cognitive Science, Rutgers University, Piscataway.

Amba Kulkarni and Devanand Shukla. 2009. *Sanskrit Morphological analyzer: Some Issues*. To appear in Bh.K Festschrift volume by LSI.

Choochart Haruechaiyasak, Sarawoot Kongyoung, and Matthew N. Dailey. 2008. *A Comparative Study on Thai Word Segmentation Approaches*. ECTI-CON, Krabi.

Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. 2007. *OpenFst: A General and Efficient Weighted Finite-State Transducer Library*. CIAA'07, Prague, Czech Republic.

Deniz Yuret and Ergun Biçici. 2009. *Modeling Morphologically Rich Languages Using Split Words and Unstructured Dependencies*. ACL-IJCNLP'09, Singapore.

DINH Q. Thang, LE H. Phuong, NGUYEN T. M. Huyen, NGUYEN C. Tu, Mathias Rossignol, and VU X. Luong. 2008. *Word Segmentation of Vietnamese Texts: a Comparison of Approaches*. LREC'08, Marrakech, Morocco.

Gérard Huet. 2009. *Formal structure of Sanskrit text: Requirements analysis for a mechanical Sanskrit processor*. Sanskrit Computational Linguistics 1 & 2, pages 266-277, Springer-Verlag LNAI 5402.

John C. J. Hoeks and Petra Hendriks. 2005. *Optimality Theory and Human Sentence Processing: The Case of Coordination*. Proceedings of the 27th Annual Meeting of the Cognitive Science Society, Erlbaum, Mahwah, NJ, pp. 959–964.

Kenneth R. Beesley. 1998. *Arabic morphology using only finite-state operations* Proceedings of the ACL Workshop on Computational Approaches to Semitic Languages, Montréal, Québec.

Leonardo Badino. 2004. *Chinese Text Word-Segmentation Considering Semantic Links among Sentences*. INTERSPEECH 2004 - ICSLP , Jeju, Korea.